Marist College
MS in Computer Science
School of Computer Science and Mathematics

CMPT-435L-111-23s Algorithm Analysis and Design
Brian Gormanly
Spring 2023

Assignment 2: Sorting

Connor H. Johnson
connor.johnson1@marist.edu

Assignment 2: Sorting

Connor H. Johnson
connor.johnson1@marist.edu

March 11, 2023

Overview

This document will cover how I created the algorithms for Selection, Insertion, Merge, and Quick sort inside a java file that successfully sorts out 666 phrases inside of a text file. I will also include the following:

- The code I used to complete this project.
- Short explanations on certain parts of the code.
- Resources to look at for reference.

Code listings

First, I would like to show you the two files I used to create this project: `mainProgram.java` and `sortingAlgorithms.java`.

The code provided will not consist of comments for visualization purposes; if you want to see my comments in my code, please visit the GitHub repository[1]

1 mainProgram.java

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.ArrayList;
4 import java.util.Scanner;
5 import sortingAlgorithms.sortingAlgorithms;
6 public class mainProgram {


7   public static void main(String[] args) {
8       try {
9           File file = new File("Lab2/textFiles/magicitems.txt");
10          Scanner scanner = new Scanner(file);
11
```

[1] GitHub: https://github.com/MaristGormanly/CJohnson-435/tree/main/Lab2

```
12          ArrayList<String> arrayList = new ArrayList<String>();
13
14          while (scanner.hasNextLine()) {
15              String line = scanner.nextLine();
16              arrayList.add(line);
17
18          }
19
20          int length = arrayList.size();
21
22          ArrayList<String> shuffledSelectionList = sortingAlgorithms.shuffle(new
ArrayList<>(arrayList), length);
23          ArrayList<String> shuffledInsertionList = sortingAlgorithms.shuffle(new
ArrayList<>(arrayList), length);
24          ArrayList<String> shuffledMergeList = sortingAlgorithms.shuffle(new
ArrayList<>(arrayList),length);
25          ArrayList<String> shuffledQuickList = sortingAlgorithms.shuffle(new
ArrayList<>(arrayList), length);
26
27          printResults(shuffledSelectionList, "Selection Sort");
28          printResults(shuffledInsertionList, "Insertion Sort");
29          printResults(shuffledMergeList, "Merge Sort");
30          printResults(shuffledQuickList, "Quick Sort");
31
32      } catch (FileNotFoundException e) {
33          e.printStackTrace();
34      }
35  }
36
37  public static void printResults(ArrayList<String> list, String sortAlgorithm) {
38      long startTime = System.nanoTime();
39      int comparisons = 0;
40      if (sortAlgorithm.equals("Selection Sort")) {
41          comparisons = sortingAlgorithms.selectionSort(list);
42      } else if (sortAlgorithm.equals("Insertion Sort")) {
43          comparisons = sortingAlgorithms.insertionSort(list);
44      } else if (sortAlgorithm.equals("Merge Sort")) {
45          comparisons = sortingAlgorithms.mergeSort(list);
46      } else if (sortAlgorithm.equals("Quick Sort")) {
47          int low = 0;
48          int high = list.size() - 1;
49          comparisons = sortingAlgorithms.quickSort(list, low, high);
50      }
51      long endTime = System.nanoTime();
52      long duration = (endTime - startTime) / 1000;
53
54      System.out.println("\n" + sortAlgorithm + ":");
55      System.out.println("\tNumber of comparisons: " + comparisons);
56      System.out.println("\tThis took: " + duration + "  s");
```

```
57  }
58 }
```

2 sortingAlgorithms.java

```
1   package sortingAlgorithms;
2   import java.util.Random;
3   import java.util.ArrayList;
4   public class sortingAlgorithms {
5
6   public static ArrayList<String> shuffle(ArrayList<String> arrayList, int length) {
7       Random random = new Random();
8
9       for (int i = length-1; i > 0; i--) {
10
11          int j = random.nextInt(i+1);
12
13          String temp = arrayList.get(i);
14          arrayList.set(i, arrayList.get(j));
15          arrayList.set(j, temp);
16      }
17      return arrayList;
18  }
19
20  public static int selectionSort(ArrayList<String> shuffledSelectionList){
21
22      int length = shuffledSelectionList.size();
23      int i = 0;
24      int comparisons = 0;
25
26      while (i < length){
27          int jMin = i;
28          int j = i + 1;
29          while (j < length){
30
31              comparisons++;
32
33              if(shuffledSelectionList.get(j).compareTo(shuffledSelectionList.get(jMi
34                  jMin = j;
35                  comparisons++;
36              }
37              j++;
38          }
39          if( jMin != i){
40              String temp = shuffledSelectionList.get(i);
41              shuffledSelectionList.set(i, shuffledSelectionList.get(jMin));
42              shuffledSelectionList.set(jMin, temp);
43
44          }
```

```
45          i++;
46      }
47
48      return comparisons;
49  }
50
51  public static int insertionSort(ArrayList<String> shuffledInsertionList){
52
53      int length = shuffledInsertionList.size();
54      int comparisons = 0;
55
56      int i = 1;
57      while (i < length){
58          int j = i;
59          while (j > 0 && shuffledInsertionList.get(j).compareTo(shuffledInsertionLis
get(j-1)) < 0){
60              comparisons++;
61
62              String temp = shuffledInsertionList.get(j);
63              shuffledInsertionList.set(j, shuffledInsertionList.get(j-1));
64              shuffledInsertionList.set(j-1, temp);
65              j--;
66          }
67          i++;
68
69      }
70
71      return comparisons;
72  }
73
74  public static int mergeSort(ArrayList<String> shuffledMergeList){
75      int comparisons = 0;
76
77      int length = shuffledMergeList.size();
78
79      if (length < 2) {
80          return comparisons;
81      }
82
83      int mid = length / 2;
84
85      ArrayList<String> left = new ArrayList<>(shuffledMergeList.subList(0, mid));
86      ArrayList<String> right = new ArrayList<>(shuffledMergeList.subList(mid, length
87
88
89      comparisons += mergeSort(left);
90      comparisons += mergeSort(right);
91
92      comparisons += merge(shuffledMergeList, left, right);
```

```
93
94      return comparisons;
95  }
96
97
98  public static int merge(ArrayList<String>shuffledMergeList,
ArrayList<String>left, ArrayList<String>right) {
99      int comparisons = 0;
100
101     int lengthL = left.size();
102     int lengthR = right.size();
103
104     int i = 0;
105     int j = 0;
106     int k = 0;
107
108     while (i < lengthL && j < lengthR) {
109         if (left.get(i).compareTo(right.get(j)) < 0) {
110             shuffledMergeList.set(k, left.get(i));
111             i++;
112             comparisons++;
113         } else {
124             shuffledMergeList.set(k, right.get(j));
125             j++;
126             comparisons++;
127         }
128         k++;
129     }
130     while (i < lengthL) {
131         shuffledMergeList.set(k, left.get(i));
132         i++;
133         k++;
134     }
135     while (j < lengthR) {
136         shuffledMergeList.set(k, right.get(j));
137         j++;
138         k++;
139     }
140     return comparisons;
141 }
142
143     public static int quickSort(ArrayList<String> shuffledQuickList, int lowI, int
144         int comparisons = 0;
145
146         if (lowI >= highI){
147             return comparisons;
148         }
149
150         int pivotIndex = highI;
```

```
151         String pivot = shuffledQuickList.get(pivotIndex);
152
153         int leftP = lowI;
154         int rightP = highI-1;
155
156         while (leftP <= rightP) {
157             while (leftP <= rightP && shuffledQuickList.get(leftP).compareTo(pivot)
158                 leftP++;
159                 comparisons++;
160             }
161             while (leftP <= rightP && shuffledQuickList.get(rightP).compareTo(pivot
162                 rightP--;
163                 comparisons++;
164             }
165             if (leftP < rightP) {
166                 String temp = shuffledQuickList.get(leftP);
167                 shuffledQuickList.set(leftP, shuffledQuickList.get(rightP));
168                 shuffledQuickList.set(rightP, temp);
169             }
170         }
171         String temp = shuffledQuickList.get(leftP);
172         shuffledQuickList.set(leftP, shuffledQuickList.get(highI));
173         shuffledQuickList.set(highI, temp);
174
175         comparisons += quickSort(shuffledQuickList, lowI, leftP - 1);
176         comparisons += quickSort(shuffledQuickList, leftP + 1, highI);
177
178
179         return comparisons;
180     }
181
182
183 }
```

Code Explanation

Now that both files can be reviewed, it is important to review parts of the code essential to completing this project. Below will be breakdowns of key components of the `sortingAlgorithms` methods being used inside of `mainProgram.java`.

1.) selectionSort() method (`sortingAlgorithms.java`)

- Inside of `sortingAlgorithms.java`, we start out by coding the selection sort (Lines: 32-62):

```
public static int selectionSort(ArrayList<String> shuffledSelectionList){

    int length = shuffledSelectionList.size();
    int i = 0;
    int comparisons = 0;

    while (i < length){
        int jMin = i;
        int j = i + 1;
        while (j < length){

            comparisons++;

            if(shuffledSelectionList.get(j).compareTo(shuffledSelectionList
            .get(jMin)) < 0){
                jMin = j;
                comparisons++;
        }
     j++;
     }
     if( jMin != i){
            String temp = shuffledSelectionList.get(i);
            shuffledSelectionList.set(i, shuffledSelectionList.get(jMin));
            shuffledSelectionList.set(jMin, temp);

    }
    i++;
    }

    return comparisons;
}
```

- the `selectionSort()` method splits the array into two parts. The sorted part at the beginning, which starts as an empty list, and the unsorted part at the end. In each iteration, the algorithm locates the smallest element in the unsorted part of the array and exchanges it with the first element of the unsorted part, adding it to the sorted part. Selection sort has a time complexity of `O(n^2)`, which makes it relatively inefficient for large arrays.

2.) insertionSort() method (`sortingAlgorithms.java`)

- Inside of `sortingAlgorithms.java` section, we construct the insertion sort method (Lines: 63-85):

```
public static int insertionSort(ArrayList<String> shuffledInsertionList){

    int length = shuffledInsertionList.size();
    int comparisons = 0;

    int i = 1;
    while (i < length){
        int j = i;
        while (j > 0 && shuffledInsertionList.get(j)
        .compareTo(shuffledInsertionList.get(j-1)) < 0){
        comparisons++; // increment comparison counter

        String temp = shuffledInsertionList.get(j);
        shuffledInsertionList.set(j, shuffledInsertionList.get(j-1));
        shuffledInsertionList.set(j-1, temp);
        j--;
    }
    i++;

    }

    return comparisons;
}
```

- the `insertionSort()` method iterates over an input list, and for each element, it finds the appropriate position within the sorted portion of the list and inserts the element there. Insertion sort has a time complexity of `O(n^2)`, which makes it relatively inefficient for large arrays. However, is effective in relation to smaller, relatively sorted arrays.

3.) mergeSort() method (`sortingAlgorithms.java`)

- Inside of `sortingAlgorithms.java` section, we construct the merge sort method (Lines: 87-148):

```
public static int mergeSort(ArrayList<String> shuffledMergeList){
    int comparisons = 0;

    int length = shuffledMergeList.size();

    if (length < 2) {
        return comparisons;
    }

    int mid = length / 2;

    ArrayList<String> left = new ArrayList<>(shuffledMergeList.subList(0, mid));
    ArrayList<String> right = new ArrayList<>(shuffledMergeList.subList(mid,
    length));
```

```
        comparisons += mergeSort(left);
        comparisons += mergeSort(right);

        comparisons += merge(shuffledMergeList, left, right);

        return comparisons;
    }


        public static int merge(ArrayList<String>shuffledMergeList,
        ArrayList<String>left, ArrayList<String>right) {
        int comparisons = 0;

        int lengthL = left.size();
        int lengthR = right.size();

        int i = 0;
        int j = 0;
        int k = 0;

        while (i < lengthL && j < lengthR) {
            if (left.get(i).compareTo(right.get(j)) < 0) {
                shuffledMergeList.set(k, left.get(i));
                i++;
                comparisons++;
            } else {
                shuffledMergeList.set(k, right.get(j));
                j++;
                comparisons++;
            }
            k++;
        }
        while (i < lengthL) {
            shuffledMergeList.set(k, left.get(i));
            i++;
            k++;
        }
        while (j < lengthR) {
            shuffledMergeList.set(k, right.get(j));
            j++;
            k++;
        }
        return comparisons;
    }
```

- the `mergeSort()` method is shown to work by recursively splitting up the array list via a left and right side. When the input is inserted into either array, it is then sorted inside its respective side. Once this is done, the algorithm then merges the two sorted halves back together to form a fully sorted list. Merge sort has a time complexity of `O(n log n)`, which makes it

relatively faster for larger lists

4.) quickSort() method (`sortingAlgorithms.java`)

- Inside of `sortingAlgorithms.java` section, we construct the quick sort method (Lines: 150-192):

```
public static int quickSort(ArrayList<String> shuffledQuickList, int lowI,
int highI){
    int comparisons = 0;

    if (lowI >= highI){
        return comparisons;
    }

    int pivotIndex = highI;
    String pivot = shuffledQuickList.get(pivotIndex);

    int leftP = lowI;
    int rightP = highI-1;

    while (leftP <= rightP) {
        while (leftP <= rightP && shuffledQuickList.get(leftP).compareTo(pivot)
        < 0) {
            leftP++;
            comparisons++;
        }
        while (leftP <= rightP && shuffledQuickList.get(rightP).compareTo(pivot)
        > 0) {
            rightP--;
            comparisons++;
        }
        if (leftP < rightP) {
            String temp = shuffledQuickList.get(leftP);
            shuffledQuickList.set(leftP, shuffledQuickList.get(rightP));
            shuffledQuickList.set(rightP, temp);
        }
    }
    String temp = shuffledQuickList.get(leftP);
    shuffledQuickList.set(leftP, shuffledQuickList.get(highI));
    shuffledQuickList.set(highI, temp);

    comparisons += quickSort(shuffledQuickList, lowI, leftP - 1);
    comparisons += quickSort(shuffledQuickList, leftP + 1, highI);


    return comparisons;
}
```

- the `quickSort()` method consists of two lists(`leftP` and `rightP`), one containing elements smaller than the pivot element and the other containing elements larger than the pivot. The two lists are then sorted recursively using a new pivot inside of the list until the entire list is sorted. Quick sort has a time complexity of `O(n log n)`, which is faster than the rest of the algorithms due to its ability to handle larger lists.

5.) shuffle() method (`sortingAlgorithms.java`)

For reference on where I found the shuffle method, please visit the site linked[2]

- Inside of `sortingAlgorithms.java` section, we construct the shuffle method (Lines: 9-28):

```
public static ArrayList<String> shuffle(ArrayList<String> arrayList, int length) {
    Random random = new Random();

    for (int i = length-1; i > 0; i--) {

        int j = random.nextInt(i+1);

        String temp = arrayList.get(i);
        arrayList.set(i, arrayList.get(j));
        arrayList.set(j, temp);
    }

    return arrayList;
}
```

- the `shuffle()` is a variation of the `O(n)` shuffle routine, Knuth shuffle. This shuffle is a commonly used shuffle that will take a random position index inside of the array and swap it with another random position index

5.) printResults() and main() methods(`mainProgram.java`)

Inside of `mainProgram.java`, we create the necessary code to present the following sorting method's number of comparisons and the duration it took µs(microseconds) (Lines: 9-62):

  - Inside of this, we are reading a specific file to 'scanning' or reading the file so that it is callable inside of our code and making new shuffled lists for each algorithm (Lines: 9-40):

```
public static void main(String[] args) {
    try {
        File file = new File("Lab2/textFiles/magicitems.txt");
        Scanner scanner = new Scanner(file);

        ArrayList<String> arrayList = new ArrayList<String>();

        while (scanner.hasNextLine()) {
```

[2] JavaTPoint: https://www.geeksforgeeks.org/shuffle-a-given-array-using-fisher-yates-shuffle-algorithm

```
            String line = scanner.nextLine();
            arrayList.add(line);

        } //end of while loop

        int length = arrayList.size();

        ArrayList<String> shuffledSelectionList = sortingAlgorithms.shuffle
        (new ArrayList<>(arrayList), length);
        ArrayList<String> shuffledInsertionList = sortingAlgorithms.shuffle
        (new ArrayList<>(arrayList), length);
        ArrayList<String> shuffledMergeList = sortingAlgorithms.shuffle
        (new ArrayList<>(arrayList),length);
        ArrayList<String> shuffledQuickList = sortingAlgorithms.shuffle
        (new ArrayList<>(arrayList), length);


        printResults(shuffledSelectionList, "Selection Sort");
        printResults(shuffledInsertionList, "Insertion Sort");
        printResults(shuffledMergeList, "Merge Sort");
        printResults(shuffledQuickList, "Quick Sort");


    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

– After we make a while loop that scans through all the lines in the .txt file found with the scanner. To make presentable print statements, we make it so that the printResults method takes in the arguments of a list, and a string that labels what algorithm was used. Inside of the method we first create the start time to get the duration, then make if-else statements to see what algorithm is being used. Finally, we make the end time and take the endTime - startTime and divide it by 1000 so it is in microseconds. After that I created 3 print statements to organize the information in a presentable way (Lines: 42-62):

```
public static void printResults(ArrayList<String> list, String sortAlgorithm)
    long startTime = System.nanoTime();
    int comparisons = 0;
    if (sortAlgorithm.equals("Selection Sort")) {
        comparisons = sortingAlgorithms.selectionSort(list);
    } else if (sortAlgorithm.equals("Insertion Sort")) {
        comparisons = sortingAlgorithms.insertionSort(list);
    } else if (sortAlgorithm.equals("Merge Sort")) {
        comparisons = sortingAlgorithms.mergeSort(list);
    } else if (sortAlgorithm.equals("Quick Sort")) {
        int low = 0;
        int high = list.size() - 1;
```

```
            comparisons = sortingAlgorithms.quickSort(list, low, high);
        }
        long endTime = System.nanoTime();
        long duration = (endTime - startTime) / 1000;

        System.out.println("\n" + sortAlgorithm + ":");
        System.out.println("\tNumber of comparisons: " + comparisons);
        System.out.println("\tThis took: " + duration + "  s");
    }
```

Resources Used

Here is a list of resources I used throughout my completion of this project:

Creating my Knuth shuffle:

- https://www.geeksforgeeks.org/shuffle-a-given-array-using-fisher-yates-shuffle-algorithm/

Algorithms for Quick sort and Merge sort:

- https://ilearn.marist.edu/access/content/group/6aa90c0a-2697-4372-a779-36abf832cc84/lecture

Algorithms for Quick sort and Merge sort:

- https://ilearn.marist.edu/access/content/group/6aa90c0a-2697-4372-a779-36abf832cc84/lecture

Nanoseconds to Microseconds conversion:

- https://www.inchcalculator.com/convert/nanosecond-to-microsecond/

Length method:

- https://www.youtube.com/watch?v=krLRbqAV6wI

Connecting java files

- https://www.youtube.com/watch?v=3ybNZM6cP3M

Debugging

- https://openai.com/blog/chatgpt/

Helping with visualization

- https://pythontutor.com/visualize.htmlmode=display